A Chemistry-Inspired Workflow Management System for Decentralizing Workflow Execution

Héctor Fernandez, Cédric Tedeschi, Thierry Priol

Abstract—With the recent widespread adoption of service-oriented architecture, the dynamic composition of services is now a crucial issue in the area of distributed computing. The coordination and execution of composite Web services are today typically conducted by heavyweight centralized workflow engines, leading to an increasing probability of processing and communication bottlenecks and failures. In addition, centralization induces higher deployment costs, such as the computing infrastructure to support the workflow engine, which is not affordable for a large number of small businesses and end-users.
In a world where platforms are more and more dynamic and *elastic* as promised by cloud computing, decentralized and dynamic interaction schemes are required. Addressing the characteristics of such platforms, nature-inspired analogies recently regained attention to provide autonomous service coordination on top of dynamic large scale platforms.
In this paper, we propose an approach for the decentralized execution of composite Web services based on an unconventional programming paradigm that relies on the chemical metaphor. It provides a high-level execution model that allows executing composite services in a decentralized manner. Composed of services communicating through a persistent shared space containing control and data flows between services, our architecture allows to distribute the composition coordination among nodes. A proof of concept is given, through the deployment of a software prototype implementing these concepts, showing the viability of an autonomic vision of service composition.

Index Terms—Service coordination, Workflow execution, Nature-inspired computing, Rule-based programming, Decentralization

1 Introduction

Loose coupling and dynamic composition are building block requirements of service oriented architectures (SOA) [1], and also two of the keys to their success. Building on these concepts, the Internet of services is now a global computing platform gathering myriads of autonomous services offering different features such as storage space, computing power, or more often software components offered to the users through the web.

SOA is now a multipurpose paradigm, facilitating business processes as well as helping scientific investigations based on compute-intensive applications. In both fields, the combinations of services allow to build more complex applications known as *composite web services* which are a temporal composition of services usually represented by a *workflow*, describing data and control dependencies between services. Recently, and in spite of the decentralized nature of the Internet, service infrastructures have built upon highly centralized architectures. Data centers and Cloud platforms act today as servers centralizing the storage and processing required for the coordination of services and, more generally, of clients (users or businesses) of the Internet. As an example, on April 2011, Amazon Elastic Compute Cloud (EC2) users experienced during 3 days an unavailability in their websites due to network problems in one of the EC2 centers, causing important losses [2]. Beyond the fact that this event showed the weaknesses of the backup services deployed by Amazon, more generally, it highlights the weaknesses of centralized architectures. Also, it forces clients to rethink their cloud strategies by relying on more decentralized solutions. Therefore, regarding the service management infrastructures, the centralized architectures lead to various weaknesses. First, they generally suffer from poor scalability and low reliability, servers being potential processing and communication bottlenecks as well as single points of failure [3]. Also, they raise privacy issues, all data and control passing through central servers and repositories.

It becomes crucial to promote a decentralized vision of service infrastructures, as for instance suggested in [4]. The benefits of a decentralized approach are manifold. First, as the processing and data are distributed among a set of nodes, there is no single point of failure. No central server acts as a potential bottleneck, network traffic is reduced, and the approach is globally more scalable. Second, the direct and asynchronous fashion of communications (without the need for central coordination) brings better throughput and graceful degradation [5]. Finally, no server takes control over data and work, each node integrating a local workflow engine (referred to as *local-engine* in the following), and having only a partial view of the composition.

More specifically, the execution of a composite Web service relies on an engine responsible for coordinating data and control flows between involved services. For the sake of illustration, let us consider a simple workflow W consisting

- *H. Fernández is with the Vrije University Amsterdam, The Netherland. E-mail: hector.fernandez@vu.nl*
- *C. Tedeschi is with the IRISA, University of Rennes 1, France. E-mail: cedric.tedeschi@inria.fr*
- *T. Priol is with INRIA, France. E-mail: thierry.priol@inria.fr*

of an activity A performed at node a followed by activity B performed at node b. In a centralized vision, during the actual execution of W, the engine first invokes A by sending a message to node a, then waits for the result of A (sent by a), and finally invokes B. With a decentralized workflow engine, nodes a and b may communicate directly (rather than through a central coordinator node) to transfer data and control when necessary (*e.g.*, after A finishes).

Over the last few years, nature-inspired metaphors have been shown to be of high interest for service coordination [6]. The *chemical programming paradigm* is a high-level execution model. Within such a model, a computation is seen as a set of reactions consuming some molecules floating and interacting freely within a *chemical solution* (close to the biological notion of *membrane*) and producing new ones. Reactions take place in an implicitly parallel, autonomous, and decentralized manner. This particular model has been shown to naturally express distributed coordination [7]. The Higher-Order Chemical Language (HOCL) [8] is a language based on these concepts and providing the higher-order: every entity in the system is seen as molecules; rules can apply to other reaction rules, opening doors to self-adaptation, the program being able to modify itself at run time. It has been shown that such a paradigm is well-suited to express service orchestration [7], and describe the enactment of workflows [9]. The proper investigation of this paper is to show that the chemical model is well-featured for underlying a decentralized execution of composite Web services and give a proof of such a concept. More precisely, the architecture proposed is decentralized in the sense that it allows each service to take part in the coordination needed to ensure the satisfaction of the (data and control) dependencies expressed in the workflow. Note that this distribution builds on top of a logically shared space. This shared space acts only as a repository with Read/Write primitives. The decentralization of this shared space is not directly tackled in this paper.

This article builds upon a previous one published in the proceedings of the ICWS conference [10]. The work presented in [10] is only conceptual and does not include any software development or experimental validation. The added value of the current article comes from the discussion of a proof of concepts and its actual deployment on top of a real platform, allowing its experimental validation. It is worth noting that the current article sums up the work conducted on the topic and thus represents a self-contained report on the subject.

The remainder of this paper is organized as follows. Section 2 presents the chemical programming paradigm in more details. Section 3 details our decentralized coordination model and language. Section 4 illustrates the work by an example of coordination of a more complex workflow. Section 5 focuses on the prototype software of the decentralized workflow engine thus designed. Section 6 details the experimental campaign and its results. Section 7 discusses similar works. Section 8 draws some conclusions.

2 The Chemical Paradigm

The chemical paradigm is a programming style based on the chemical metaphor. Molecules (data) are floating in a chemical solution, and react according to reaction rules (program) to produce new molecules (resulting data). Reactions are conditional, and take place between some molecules satisfying a reaction condition. This process continues until no more reactions can be performed: the solution is said to be *inert*. Reactions take place in an implicitly parallel and autonomous way (independently from each other), and in a non-deterministic order.

Formally, the solution is represented by a multiset containing molecules, and rewriting/transformation rules specify the reactions between molecules. The Gamma model (General Abstract Model for Multiset Manipulation) [11] has been a pioneer work realizing the chemical paradigm. The multiset, which is the formal representation of the chemical solution, is the unique data structure in Gamma. The multiset works similarly to a shared address space on which multiple processors can operate independently, applying the rules concurrently.

In this paper, we use a chemical language enhanced with higher order, called HOCL (*Higher Order Chemical Language*) [8]. In HOCL, every entity is a molecule, including reaction rules. A program is a solution of molecules, that is to say, a multiset of atoms (A_1, ..., A_n) which can be constants (integers, booleans, *etc*.), sub-solutions (denoted $\langle M_i \rangle$), or reaction rules.

Following the chemical paradigm, the execution of an HOCL program consists in applying reactions until the solution becomes inert. A reaction involves a reaction rule **one** P **by** M **if** V and a molecule N that satisfies the pattern P and the reaction condition V. The reaction consumes the rule and the molecule *N*, and produces *M*. The basic **one** P **by** M **C** reaction rule is one-shot: it disappears when it reacts. Its variant **replace** P **by** M **C** is n-shot: it is not consumed when it reacts. In the following, we use a more advanced syntax to declare and name molecules: **let** x = M_1 **in** M_2 is equivalent to M_2 where all occurrences of x are replaced by M_1. For instance, consider the following solution *MaxNumbers* which calculates the maximum value of a given set of numbers. The below example illustrates the expressiveness and higher order of HOCL, where reactions consume and/or produce other reaction rules.

$$\textbf{let } max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in } \langle 2, 3, 5, 8, 9, max \rangle$$

The rule max reacts with two integers x and y such that $x \geq y$ and replaces them by x (keep the integer with highest value). Initially, several reactions are possible: max can react with any couples of integers satisfying the condition: 2 and 3, 2 and 5, 8 and 9, *etc*. In order for the final solution to contain only the result, we introduce a higher-order rule responsible to delete the max rule once the solution only contain the highest integer value. This introduces the need for the sequentiality of events: we need to wait that all possible reactions between max and couples of integers took place before deleting the rule. Within the chemical model, the sequentiality is achieved through sub-solutions: to access a sub-solution, a rule has to wait for its inertia. In our example, this leads to the encapsulation of the solution:

$$\langle\langle 2,3,5,8,9,max\rangle,\ \textbf{one}\ \langle max = m, \omega\rangle\ \textbf{by}\ \omega\rangle$$

The m variable matches a rule named max, and ω matches all the remaining elements. One possible execution scenario within the sub-solution is the following (2 and 8, as well as 3 and 5, react first, producing the intermediate state):

$$\langle 2,3,5,8,9,max\rangle \rightarrow^* \langle 3,5,9,max\rangle \rightarrow^* \langle 9,max\rangle$$

Once the inertia is reached within the sub-solution, the one-shot rule can be triggered, extracting the result:

$$\langle\langle 9,max\rangle,\ \textbf{one}\ \langle max = m, \omega\rangle\ \textbf{by}\ \omega\rangle \rightarrow \langle 9\rangle$$

As we illustrated with a fine-grain example, HOCL provides the ability to express autonomic coordination of rules (without the need for any centralized control). The current state of a computation is represented by the solution, that constitutes an information system by itself. In other words, the multiset is a shared space providing the information required for dynamic coordination, such as a decentralized workflow execution.

3 Chemical Decentralized Workflow Execution

In this section, we describe our decentralized architecture for workflow coordination based on a higher-order *chemical* framework, illustrating the adequacy of the chemical paradigm to execute composite Web services.

3.1 Architecture

As illustrated by Figure 1, the proposed architecture is composed by two core elements, namely the **Chemical Web Service (ChWS)** and the **multiset**. A ChWS is a chemical encapsulation of a Web service. It is co-responsible with other ChWSes of the coordination of the execution of workflows. Physically, ChWSes are hosted by some nodes and logically identified by symbolic names into the multiset. Each ChWS is basically equipped with three elements, namely:

1) The *service caller* represents the encapsulation of a Web service invocation. The invocation, to an effective possibly distant Web service, is encapsulated in a chemical expression readable by a chemical interpreter. The implementation of the Web service itself is not encapsulated, as shown in Figure 1.
2) A local storage space containing part of the multiset, *i.e.*, molecules and reaction rules constituting the data and control dependencies related to the coordination of the workflow execution.
3) An HOCL interpreter, working as the chemical local-engine executing the reactions according to molecules and reaction rules stored in the multiset, responsible for applying the defined workflow patterns and transferring data and control information to other ChWSes involved in a workflow.

The multiset acts as a space shared by all ChWSes involved in the workflow. It contains the workflow definition and all information needed by ChWSes for a decentralized execution of a workflow, and in which each ChWS can operate independently. This information combines molecules representing data and ChWSes, rules representing control dependencies of the workflow, and rules for the coordination of its execution, as illustrated by Figure 2. Data and control dependencies of the workflow are defined beforehand using some workflow executable languages, like the well-known BPEL [12], an XML-based workflow language for Web services, or any other workflow language. For instance, a BPEL specification could be translated into a chemical program, as detailed in Section 3.2. Even though HOCL is used to describe and execute workflow specifications, our purpose is to show its potential as an executable workflow language, as detailed in [13]. To coordinate the execution of the workflow, we also need some additional chemical rules, which are *generic*, *i.e.*, independent of a specific workflow. Section 3.3 focuses on these generic rules.

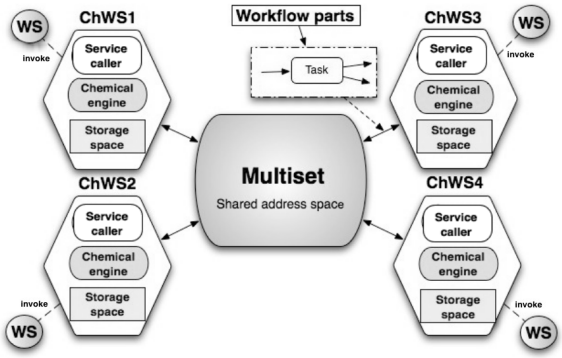

Fig. 1. The proposed architecture.

The multiset shares some conceptual similarities with the Distributed Shared Memory (DSM) paradigm [14], developed in the area of distributed operating systems. DSM maps a globally unique logical memory address to a local physical memory slot, thus emulating a shared global space on top of a distributed memory platform. By analogy, multiset mirrors DSM's behavior by exposing molecules and reactions rules physically scattered across a set of ChWSes in a single shared space.

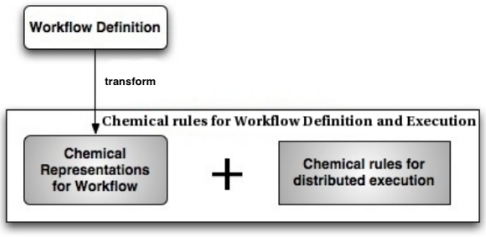

Fig. 2. Chemical workflow.

In other words, from a conceptual point of view (illustrated by Figure 1), ChWSes communicate through a unique global multiset containing all information needed by ChWSes to execute their part of a workflow. ChWSes exchange data and control dependencies through this multiset. In a classical centralized workflow architecture, the services themselves do not know these dependencies, as an engine manages all information and coordinates the whole execution.

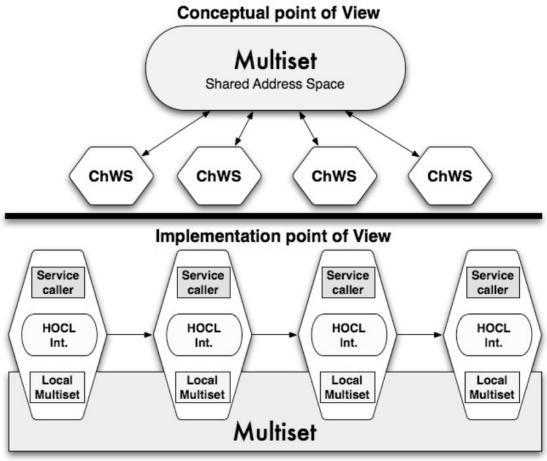

Fig. 3. Points of view of the architecture.

From an implementation point of view, the multiset is physically distributed. While apparently, each ChWS only interacts with the multiset, physically, data and control information (molecules and reaction rules of the multiset) are effectively transferred between local storages of ChWSes. Put together, the molecules stored by ChWS form the multiset. Figure 3 summarizes these two points of view: the upper side shows the conceptual point of view where all ChWSes are *connected* through one multiset; the lower part shows the implementation point of view where all ChWSes are directly interconnected through the multiset, the reactions and molecules being directly transferred from one ChWS to another one using a distributed multiset. In this paper, we will assume that distributing the shared space is possible and will focus on decentralizing the coordination processing itself. Please refer to [15] for more information on how to distribute the shared space. Figure 3 provides a simple example where all ChWS are connected through a sequential workflow (modeled by arrows), but any workflow pattern could be modeled, as we detailed in our previous work [16].

3.2 Chemical Workflow Representation

In order to express all data and control dependencies of a workflow definition according to the chemical paradigm and to distribute the information among ChWSes, we use a series of chemical abstractions inspired by the work in [9]. These abstractions allow representing a workflow definition with the HOCL language. Such a representation is given in Figure 6.

As a chemical expression, the whole solution represents the multiset containing all information. The solution itself is composed of as many sub-solutions as ChWSes. Each sub-solution represents a ChWS with its data and control dependencies with other ChWSes within the workflow definition. More formally, a ChWS is one molecule of the form $ChWSi : \langle \dots \rangle$ where ChWSi refers to the symbolic name given to the service whose connection details and physical position are hidden, as shown in Figure 6.

```
1.01  ⟨ // Multiset (Solution)
1.02        ChWSi:⟨. . . ⟩ // ChWS (Sub-solution)
1.03        ChWSi+1:⟨. . . ⟩
1.04        . . .
1.05        ChWSn:⟨ . . . ⟩
1.06  ⟩
```

Fig. 4. Chemical workflow representation

Let us consider a simple workflow expressed using BPMN (Business Process Modeling Notation) [17], and composed of the four services S_1, S_2, S_3 and S_4, as illustrated in Figure 5. In this example, after S_1 completes, S_2 and S_3 can be invoked in parallel. Once S_2 and S_3 have both completed, S_4 can be invoked.

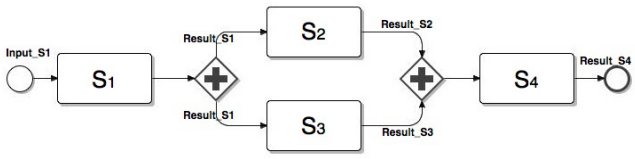

Fig. 5. Simple workflow example.

The corresponding chemical representation for this workflow is presented in Figure 6. As we already mentioned, the solution contains as many sub-solutions as Web services. $ChWS1 : \langle \dots \rangle$ to $ChWS4 : \langle \dots \rangle$ represent ChWSes in the solution. The relations between ChWSes are expressed through molecules of the form DEST*:ChWSi* with $ChWSi$ being the destination ChWS where some information needs to be transferred. For instance, we can see in $ChWS1$ sub-solution that ChWS1 will transfer some information (the outcome of ChWS1) to ChWS2 and ChWS3 (Line 2.02).

Let us focus on the details of these dependencies. ChWS2 has a *data* dependency: it requires a molecule RESULT*:ChWS1:value1* containing the result of S_1 to be invoked (second part of Line 2.04). The two molecules produced by the reaction represent the call to S_2 and their input parameters. They are expressed using a molecule of the form CALL*:Si*, and a molecule PARAM:$\langle in_1,...,in_n \rangle$, where in_1, ..., in_n represent the input parameters to call the service Si. In Figure 6, this input parameter corresponds to the result of some previous service S_j. ChWS3 works similarly.

Occasionally, on a particular service, data and control dependencies may differ. Consider ChWS4. As specified by Figure 6, ChWS4 needs to wait until ChWS2 and ChWS3 have been completed. This constitutes a control dependency known as *synchronization*. However, as we can see in line 2.08, the

```
2.01  ⟨
2.02  ChWS1:⟨DEST:ChWS2, DEST:ChWS3⟩,
2.03  ChWS2:⟨DEST:ChWS4, replace RESULT:ChWS1:value1
2.04                     by CALL:S2, PARAM:⟨(value1)⟩ ⟩,
2.05  ChWS3:⟨DEST:ChWS4, replace RESULT:ChWS1:value1
2.06                     by CALL:S3, PARAM:⟨(value1)⟩ ⟩,
2.07  ChWS4:⟨replace RESULT:ChWS2:value2, RESULT:ChWS3:value3
2.08           by CALL:S4, PARAM:⟨(value2)⟩
2.09  ⟩
```

Fig. 6. Chemical representation for the workflow of Figure 5

service S_4 is invoked only on $value2$ which is the result of S_2. This constitutes a data dependency. The $ChWS4$ sub-solution contains one reaction rule translating those dependencies in chemical language (see line 2.08): the presence of molecules RESULT*:ChWS2:value2* and RESULT*:ChWS3:value3* inside the $ChWS4$ sub-solution expresses the fulfillment of the control dependencies, to start its own execution. In addition, a data dependency is also expressed in ChWS4: the result of S_2 is required to call S_4. During the execution, as soon as RESULT*:ChWS2:value2* and RESULT*:ChWS3:value3* appear in the $ChWS4$ sub-solution, the local engine of ChWS4 will be able to perform the reaction that will produce two new molecules of the form CALL:S4 and PARAM:⟨ (value2) ⟩ to call the effective service S_4 on the input $value2$.

To sum up, one reaction rule can express both control and data dependencies. In contrast with the previous synchronization pattern, the simple data dependencies are enough to express the parallel split pattern of S_1 with S_2 and S_3. Thanks to the implicit parallelism of the chemical execution model, the reaction rules inside $ChWS2$ and $ChWS3$ can be executed in parallel. Therefore, ChWS2 and ChWS3 will receive the result of S_1 from ChWS1 and the invocation of S_2 and S_3 will take place in parallel.

This fragment of HOCL code is the chemical representation of a workflow, that will be interpreted by chemical local engines, performing the decentralized execution of this workflow thanks to a set of generic rules we introduce in the next sections.

3.3 Generic Rules for Invocation and Transfer

As previously mentioned, to ensure the execution of a chemical workflow, additional chemical *generic* rules (i.e., independent of any workflow) must be defined. These rules are included in the chemical local engines and are responsible for the efficient execution of the workflow. We now review three of these *generic* rules, illustrated in Algorithm 1, responsible for these tasks, and that will be commonly encountered in the compositions presented later. The *invokeServ* rule encapsulates the actual invocation of services. When reacting, it invokes the Web Service S_i, by consuming the tuples CALL*:Si* representing the invocation itself, and PARAM*:⟨in$_1$,...,in$_n$⟩* representing its input parameters, and generates the molecules containing the results of the invocation in the ChWSi sub-solution. The molecule FLAG_INVOKE is a flag whose presence in the solution indicates that the invocation can take place. The *preparePass* rule is used for preparing the messages to transfer the results to their destination services, that will later trigger the execution of the *passInfo* rule. Thus, the *preparePass* rule captures one molecule of the form *ChWSi:*⟨RESULT*:ChWSi:*⟨*value*⟩, DEST*:ChWSj,* ω ⟩. RESULT*:ChWSi:*⟨*value*⟩ is the result of S_i, while DEST*:ChWSj* comes from the chemical specification of the workflow such as the one presented in Figure 6.

Algorithm 1 Basic generic rules.

```
3.01  let invokeServ = replace ChWSi:⟨CALL:Si, PARAM:⟨in_1, ..., in_n⟩,
3.02                                FLAG_INVOKE, ω ⟩,
3.03                   by ChWSi:⟨RESULT:ChWSi:⟨value⟩, ω ⟩
3.04  let preparePass = replace ChWSi:⟨RESULT:ChWSi:⟨value⟩, DEST:ChWSj, ω⟩
3.05                   by ChWSi:⟨PASS:ChWSj:⟨COMPLETED:ChWSi:⟨value⟩ ⟩, ω⟩
3.06  let passInfo = replace ChWSi:⟨PASS:ChWSj:⟨ ω_1 ⟩, ω_2 ⟩, ChWSj:⟨ ω_3 ⟩
3.07                   by ChWSi:⟨ ω_2 ⟩, ChWSj:⟨ ω_1, ω_3 ⟩
```

Rule *passInfo* transfers molecules of information between ChWSes. This rule reacts with a molecule *ChWSi:*⟨PASS*:d:*⟨ω_1 ⟩⟩ that indicates that some molecules (here denoted ω_1) from ChWSi needs to be transferred to d. These molecules, once inside the sub-solution of d will trigger the next step of the execution. Therefore, the molecule ω_1 will be transferred from sub-solution *ChWSi* to sub-solution *ChWSj*, when reacting with *passInfo* rule.

Thanks to these reaction rules, the execution of a chemical workflow is decentralized since each ChWS is able to execute rules using its embedded HOCL interpreter, each ChWS achieving the coordination related to the service it encapsulates. However, they can not, by themselves, solve how to *distribute the workflow patterns responsibilities among participants*[1]. Accordingly, we defined a set of generic rules for solving complex workflow pattern, as detailed in our work [13]. We do not include them here, as the chemical definition of complex workflow structures is not our main concern in this paper.

4 EXECUTION EXAMPLE

To better understand how the coordination between chemical engines works, we here present the execution of the workflow example illustrated in Figure 6, for which we focus on each step of the coordination logic. These steps are listed in Figures 7 (steps 1-3), 8 (steps 4-7) and 9 (steps 8-10). Recall that, thanks to the higher-order property, reaction rules react themselves with other molecules. As we have discussed already, the example is composed by four ChWSes applying *parallel split* and *synchronization* patterns. The execution is as follows: After *ChWS1* completes, it forwards the result to *ChWS2* and *ChWS3* in parallel. Once *ChWS2* and *ChWS3* have completed, *ChWS4* can start. Consider that each chemical local

1. Each ChWS is seen as a participant in a workflow definition.

engine is responsible for the reactions taking place within its sub-solution in the multiset, thus respecting at runtime the decentralization designed. Indeed, for the sake of clarity, we only mention the molecules that take part in the logic of the coordination.

The first step (Lines 4.02-4.05) corresponds to the initial state of the multiset, illustrated in Figure 7. Initially, the only possible reaction is inside *ChWS1*, the *invokeServ* rule is triggered by the HOCL interpreter of *ChWS1*, producing the outcome molecule RESULT*:ChWS1:*⟨*val*⟩. This molecule represents the result of the invocation of S_1. Then, the *preparePass* rule consumes the molecules DEST*:destination* and RESULT*:ChWS1:*⟨*val*⟩, preparing the *parallel split*. Therefore, it produces two new molecules for the distribution of this result to *ChWS2* and *ChWS3* (Line 4.20). Finally, still through *ChWS1*, *passInfo* triggers it by transferring in parallel the outcome of *ChWS1*.

Once the information is received by *ChWS2* and *ChWS3*, the reactions (Lines 5.04 and 5.06) are triggered, in parallel, producing the needed molecules to invoke S_2 and S_3. Thus, molecules of the form CALL:Si and PARAM:(val) contained into *ChWS2* and *ChWS3* respectively, launch the *invokeServ* rule (Lines 5.03-5.05) that generates the result of S_2 and S_3. Similarly to *ChWS1*, the molecules RESULT*:ChWS2:*⟨*val2*⟩ and RESULT*:ChWS3:*⟨*val3*⟩ react with the *preparePass* rule. Finally, in *ChWS2* and *ChWS3*, the *passInfo* rule propagates the molecule PASS*:ChWS4:*⟨ *information* ⟩ to *ChWS4* (Lines 5.23-5.24).

The execution ends with steps in Figure 9, processed by ChWS4's local engine. Once the information from *ChWS2* and *ChWS3* is received by *ChWS4*, the reaction rule (Line 6.06) can react with results molecules to produce two new molecules for invoking service S_4 (Line 6.12). Finally, *invokeServ* rule will take place producing the final result RESULT*:ChWS4:*⟨*val4*⟩.

With this example, we have shown that local engines within ChWSes are co-responsible for applying workflow patterns, invoking services, and propagating the information to other ChWSes. The coordination is achieved as reactions become possible, in an asynchronous and decentralized manner.

5 SOFTWARE PROTOTYPE

To put in practice and validate the concepts presented, we have developed an architectural framework and two software prototypes exhibiting different processing and communication techniques. Firstly, we developed a *decentralized*, shared space-based architecture inspired by that presented in Section 3. This architecture is composed of a set of chemical engines collaborating through a multiset acting as a shared space. Service interactions are loosely coupled, a property inherited by the adoption of the tuplespace model. The computation is decentralized (even if the multiset remains a centralized.) Secondly, and for the sake of comparison and discussion, we also developed a centralized architecture, which is composed of a unique chemical engine playing the same role as traditional workflow engines. Both prototypes are written in Java and built atop an HOCL interpreter based on *on-the-fly* compilation of HOCL specifications [18]. The table below summarizes them:

```
4.01 ⟨
4.02 ChWS1:⟨DEST:ChWS2,DEST:ChWS3, invokeServ, preparePass, passInfo,
       CALL:S1, PARAM:in1⟩,
4.03 ChWS2:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S2, PARAM:(val)⟩,
4.04 ChWS3:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S3 PARAM:(val)⟩,
4.05 ChWS4:⟨invokeServ,
       replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
       by CALL:S4, PARAM:(val2)⟩
4.06 ⟩
                              ↓
4.07 ⟨
4.08 ChWS1:⟨DEST:ChWS2,DEST:ChWS3, preparePass, passInfo, invokeServ,
       CALL:S1, PARAM:in1 ⟩,
4.09 ChWS2:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S2, PARAM:(val)⟩,
4.10 ChWS3:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S3 PARAM:(val)⟩,
4.11 ChWS4:⟨invokeServ,
       replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
       by CALL:S4, PARAM:(val2)⟩
4.12 ⟩
                              ↓
4.13 ⟨
4.14 ChWS1:⟨DEST:ChWS2,DEST:ChWS3, preparePass, passInfo,
       RESULT:ChWS1:⟨val⟩⟩,
4.15 ChWS2:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S2, PARAM:(val)⟩,
4.16 ChWS3:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S3 PARAM:(val)⟩,
4.17 ChWS4:⟨invokeServ,
       replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
       by CALL:S4, PARAM:(val2)⟩
4.18 ⟩
                              ↓
4.19 ⟨
4.20 ChWS1:⟨passInfo, PASS:ChWS2:⟨COMPLETED:ChWS1:⟨val⟩ ⟩,
       RESULT:ChWS1:⟨val⟩, PASS:ChWS3:⟨COMPLETED:ChWS1:⟨val⟩ ⟩ ⟩,
4.21 ChWS2:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S2, PARAM:(val)⟩,
4.22 ChWS3:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
       replace COMPLETED:ChWS1:⟨val⟩ by CALL:S3 PARAM:(val)⟩,
4.23 ChWS4:⟨invokeServ,
       replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
       by CALL:S4, PARAM:(val2)⟩
4.24 ⟩
```

Fig. 7. Workflow execution, steps 1-3.

5.01 ⟨
5.02 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
5.03 ChWS2:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
COMPLETED:ChWS1:⟨val⟩,
5.04 **replace** COMPLETED:ChWS1:⟨val⟩ **by** CALL:S2, PARAM:(val)⟩,
5.05 ChWS3:⟨DEST:ChWS4, invokeServ, preparePass, passInfo,
COMPLETED:ChWS1:⟨val⟩,
5.06 **replace** COMPLETED:ChWS1:⟨val⟩ **by** CALL:S3, PARAM:(val)⟩,
5.07 ChWS4:⟨invokeServ,
replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
by CALL:S4, PARAM:(val2)⟩
5.08 ⟩
↓
5.09 ⟨
5.10 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
5.11 ChWS2:⟨DEST:ChWS4, **invokeServ**, preparePass, passInfo,
CALL:S2, PARAM:(val)⟩,
5.12 ChWS3:⟨DEST:ChWS4, **invokeServ**, preparePass, passInfo,
CALL:S3, PARAM:(val)⟩,
5.13 ChWS4:⟨invokeServ,
replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
by CALL:S4, PARAM:(val2)⟩
5.14 ⟩
↓
5.15 ⟨
5.16 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
5.17 ChWS2:⟨**DEST:ChWS4, RESULT:ChWS2:⟨val2⟩, preparePass**, passInfo⟩,
5.18 ChWS3:⟨**DEST:ChWS4, RESULT:ChWS3:⟨val3⟩, preparePass**, passInfo⟩,
5.19 ChWS4:⟨invokeServ,
replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
by CALL:S4, PARAM:(val2)⟩
5.20 ⟩
↓
5.21 ⟨
5.22 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
5.23 ChWS2:⟨**PASS:ChWS4:⟨COMPLETED:ChWS2:⟨val2⟩ ⟩, passInfo**,
RESULT:ChWS2:⟨val2⟩ ⟩,
5.24 ChWS3:⟨**PASS:ChWS4:⟨COMPLETED:ChWS3:⟨val3⟩ ⟩, passInfo**,
RESULT:ChWS3:⟨val3⟩ ⟩,
5.25 ChWS4:⟨invokeServ,
replace COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
by CALL:S4, PARAM:(val2)⟩
5.26 ⟩

Fig. 8. Workflow execution, steps 4-7.

6.01 ⟨
6.02 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
6.03 ChWS2:⟨RESULT:ChWS2:⟨val2⟩ ⟩,
6.04 ChWS3:⟨RESULT:ChWS3:⟨val3⟩ ⟩,
6.05 ChWS4:⟨invokeServ,**COMPLETED:ChWS2:⟨val2⟩**,**COMPLETED:ChWS3:⟨val3⟩**,
6.06 **replace** COMPLETED:ChWS2:⟨val2⟩, COMPLETED:ChWS3:⟨val3⟩
by CALL:S4, PARAM:(val2)⟩
6.07 ⟩
↓
6.08 ⟨
6.09 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
6.10 ChWS2:⟨RESULT:ChWS2:⟨val2⟩ ⟩,
6.11 ChWS3:⟨RESULT:ChWS3:⟨val3⟩ ⟩,
6.12 ChWS4:⟨**invokeServ, CALL:S4, PARAM:(val2)**⟩
6.13 ⟩
↓
6.14 ⟨
6.15 ChWS1:⟨RESULT:ChWS1:⟨val⟩ ⟩,
6.16 ChWS2:⟨RESULT:ChWS2:⟨val2⟩ ⟩,
6.17 ChWS3:⟨RESULT:ChWS3:⟨val3⟩ ⟩,
6.18 **ChWS4:⟨RESULT:ChWS4:⟨val4⟩** ⟩
6.19 ⟩

Fig. 9. Workflow execution, steps 8-10.

	Centralized Arch.	**Decentralized Arch.**
Execution	Centralized	Decentralized
Multiset	Centralized	Centralized
Communication	Loosely coupled	Loosely coupled

5.1 Centralized version

Following the examples of most of workflow management systems, the coordination can be managed by single node, referred to as the *chemical workflow service*, as illustrated by Figure 10.

As mentioned in Section 3, the workflow definition is executed as a chemical program by the chemical workflow service. The low layer of the architecture is an HOCL interpreter. Given a workflow specification as input (an HOCL program), it executes the workflow coordination by reading and writing the multiset initially fed with the workflow definition. The interface between the chemical engine and the distant services themselves is realized through the *service caller*. The service caller relies on the DAIOS framework [19], which provides an abstraction layer allowing the dynamic connection to different flavors of services (SOAP or RESTFul), abstracting the target service's internals. DAIOS was specially extended with a module which automatically generates dynamic bindings, as well as input and output messages required between the chemical engine and a Web service.

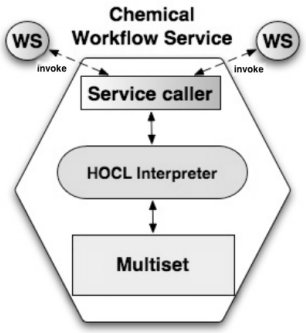

Fig. 10. Centralized architecture.

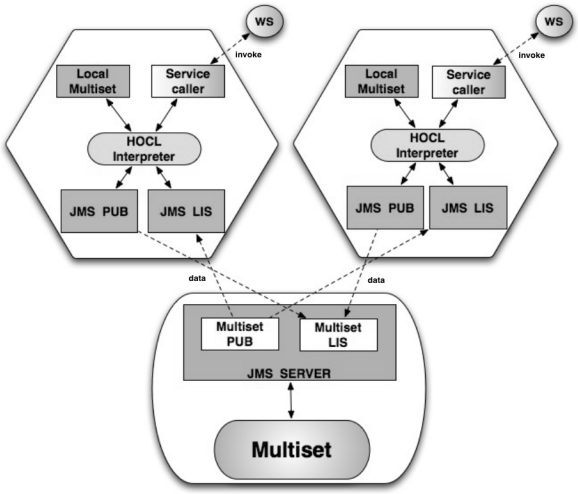

Fig. 11. Decentralized architecture.

5.2 Decentralized version

This framework is similar to the previous architecture, however the functionality of the multiset represents the main difference with the centralized version, as illustrated by Figure 11. The multiset is initially fed with the HOCL specification of the workflow. The multiset acts as a shared space playing the role of a communication medium and a storage system, while each ChWS involved will take its part in the coordination process. As such, the coordination workload [2] is now distributed among the ChWSes participating in the workflow.

As we detailed in Section 3.2, the workflow definition is comprised of one sub-solution per WS involved; the information in one sub-solution can only be accessed by the ChWS owner of/represented by that sub-solution. On each ChWS, a local storage space acts as a temporary container for the sub-solution to be processed by the local HOCL interpreter. The interface between a ChWS and a concrete WS is still realized through the *service caller* based on the DAIOS framework. ChWSes communicate with the multiset through the Java Message Service (JMS) publisher/subscriber modules. The multiset is encapsulated into a JMS server to allow concurrent reading and writing operations by ChWSes. Periodically, and independently from each other, ChWSes read their sub-solution from the multiset. The sub-solution obtained is then locally processed by the ChWS's local HOCL interpreter and then pushed back to the multiset for update.

This architecture follows a loosely coupled interaction model, as ChWSes only keep a reference to the shared space, instead of having a reference to each ChWS with which they interact.

5.2.1 Communications

As mentioned above, communication mechanisms are implemented with JMS. JMS modules are included into the ChWSes, and the multiset is a JMS server.

The publish/subscribe messaging model is used by the ChWSes and the multiset whereby message producers called publishers pushing each message to each interested party called subscribers. Initially, the *Multiset PUBlisher* pushes the content of each WSi solution to each *ChWSes LIStener*. On the ChWS's side, the *ChWS LIStener* receives the content of the ChWSi solution which will be copied into its local multiset. Once the HOCL interpreter is done with its execution, the *ChWS PUBlisher* pushes the content of its sub-solution into the *Multiset LIStener*.

Recall that the decentralized architecture is distributed, a JMS server into the multiset is needed to coordinate all these messages. Concretely, we use *ActiveMQ* (version 5.4.1) an implementation of the JMS 1.1 specification, which can be embedded in a Java application server. This ActiveMQ server allows to register and save all the message exchanges between subscribers and publishers. The exchanged messages are stored in the server, allowing them to be used in the future if a problem arises during the transaction.

6 Experimental Results

Our objective is here to better capture the behavior of a decentralized chemistry-based workflow system. To achieve it, we processed workflows with different characteristics using our centralized and decentralized prototypes. These characteristics are the number of tasks involved, the amount of data exchanged and the complexity of the coordination required [3].

2. The coordination workload includes all the workflow operations related with the processing of workflow structures.

3. Informally, we consider as a complex workflow, a workflow having *many* patterns to be applied and a high rate of data exchange. Workflows are more precisely described in the following.

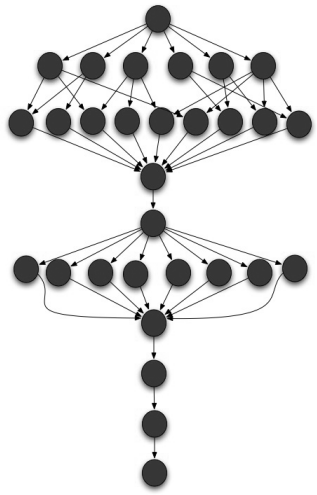

Fig. 12. 30-task graph.

Experiments were conducted over the nation-wide Grid'5000 platform [20]. More specifically, these experiments were conducted on the *parapide, paramount and paradent* clusters, located in Rennes. The *parapide* cluster is composed of nodes equipped with two quad-core Intel Xeon X5570, 24 GB of RAM; the *paramount* cluster provides nodes with two quad-core Intel Xeon L5148 LV processors, 30 GB of RAM, and the *paradent* cluster is equipped with two quad-core Intel Xeon L5420 processors. All three clusters are furnished with 40GB InfiniBand Ethernet cards.

6.1 Workflows Considered

Three workflows containing 30, 60 and 100 tasks were designed inspired by the graph of the Montage workflow [21], a classic astronomical image mosaic workflow processing large images of the sky. Montage combines sequential and parallel flows, making it relevant for such experiments. Our variants of the Montage workflow are illustrated in Figure 12, Figure 13 and Figure 14, and are respectively referred to as *Workflow30t*, that comprises 30 tasks over 10 *levels* (the level of a task is defined as the length of the path leading to it from the source task),*Workflow60t*, that comprises 60 tasks dispatched over 13 levels, and *Workflow100t* made of 100 tasks of 19 levels. Our campaign has the following considerations:

1) Each task calls an actual web service.
2) Tasks at the same level have the same computational cost.
3) The results of these experiments are averaged over 10 runs.
4) Each task is run by one distinct machine on the Grid'5000 platform.

Three different web services were built, needing different amounts of data exchanges, for one call to this service, namely 28 bytes for *serviceA*, 583 bytes for *serviceB*, and 3063 bytes for *serviceC*. The definitions used for each workflow are available online[4].

4. https://www.irisa.fr/myriads/members/hfernand/hocl/workflowsJournal

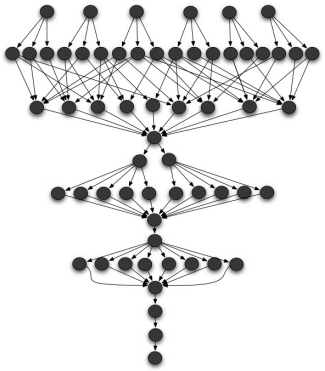

Fig. 13. 60-task graph.

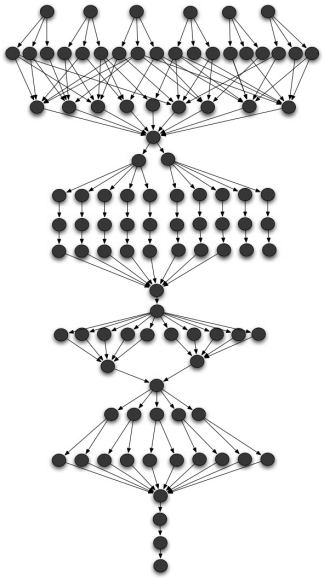

Fig. 14. 100-task graph.

6.2 Managing Large Workflows

In this experiment, we utilized our centralized and decentralized prototypes to process the *Worflow30t, Workflow60t* and *Workflow100t*. To create a more heterogeneous scenario, we also decided to repeat these workflows using the *serviceA, serviceB and serviceC*.

When looking at the results shown on Figures 15, 16, and 17, we can draw several conclusions. Firstly, the size of each of these workflows affects its execution time. The *Workflow30t* presents lower execution times than the *Workflow60t*, and *Workflow100t*, independently of the workflow management systems utilized. Secondly, the difference of performance between the centralized and decentralized engines is very low when *serviceA* is used. Then, when shifting to services with a higher amount of data exchanged, the performance of the centralized approach drops significantly. Finally, when using the decentralized workflow system, the *Workflow60t* and *Workflow100t* presents an important improvement in the performance.

In contrast, our centralized workflow system shows a sub-

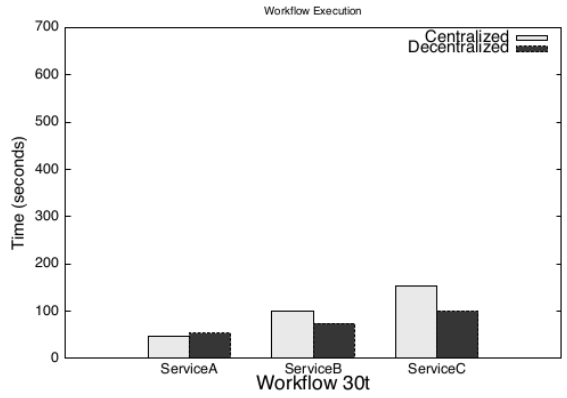

Fig. 15. Execution times for the 30-task graph.

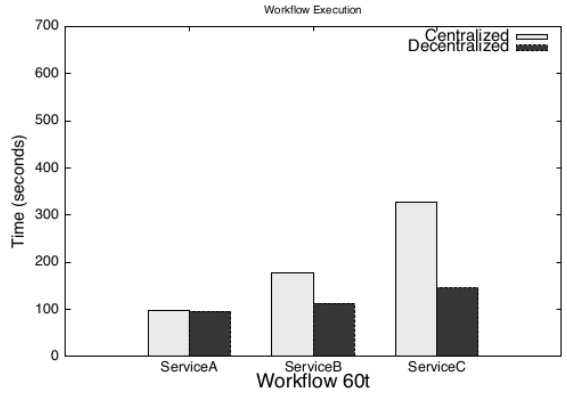

Fig. 16. Execution times for the 60-task graph.

stantial increase of the execution time caused by the two aforementioned factors: the number of tasks to be coordinated, and the type of service. This suggests that the efficiency of an engine is affected by the amount of data exchanged among services participating in a workflow. Obviously, a central engine is in charge of the management of all data exchanged between services, increasing the processing time of a workflow.

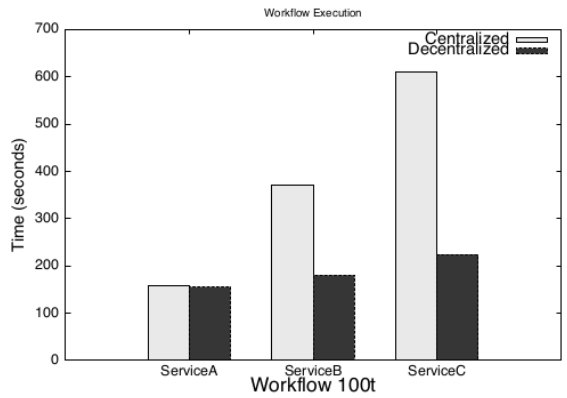

Fig. 17. Execution times for the 100-task graph.

6.2.1 Coordination Workload

We now focus on the results of each workflow using the centralized engine in Figure 18. Here, we extracted from the total execution time, the cost in time of the coordination activities (denoted by *Coordination*), and the execution time required by the web services to return the results of their invocations (denoted by *Service invocation*). Informally, we consider as *coordination activities*, all the operations managing the data exchanged among tasks and applying the different workflow patterns. Consequently, the coordination time will increase depending on the number and complexity of patterns to be applied and the rate of data exchanged.

As observed in Figure 18, the processing time spent in coordination activities represents the majority of the total execution time in all the workflows. This phenomenon shows the importance of the engines to efficiently handle workflows with a substantial coordination workload. As an example, the centralized engine seems saturated when processing the coordination activities of the *Workflow100t* using the *serviceC*. Furthermore, this graph also shows similar execution times regarding the service invocation in each workflow (independently of the type of service), so that these times can be omitted in the following, as they mostly remain constant for both prototypes. Therefore, we focus on the coordination time employed for each prototype to process the different workflows, as it allows us to identify the benefits behind the decentralized workflow execution.

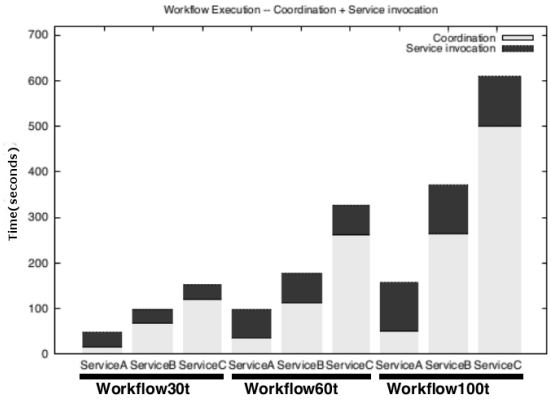

Fig. 18. Centralized execution: ratio between coordination and invocation.

The coordination times employed by both centralized and decentralized workflow systems are now shown in Figures 19, 20 and 21. It shows a reduction in the coordination times spent by the decentralized system in comparison to the centralized one. This improvement in the performance is achieved due to the coordination workload is distributed among the different engines, reducing the final execution time. In contrast to that of a centralized workflow engine, the coordination workload is managed by a single node, thus explaining the increment in the coordination time experienced for the workflows with the *serviceB* and *serviceC*. As we mentioned before, the benefits behind the decentralized workflow execution can be observed when processing the *Workflow60t* and

Workflow100t using the *serviceB* and *serviceC*, respectively. However, the use of the *serviceA* in the workflows do not offer any gain due to its simplicity, the reduced computational load of this workflow provokes that the coordination time is higher than or equal to that of the centralized engine, due to the communications (network latency).

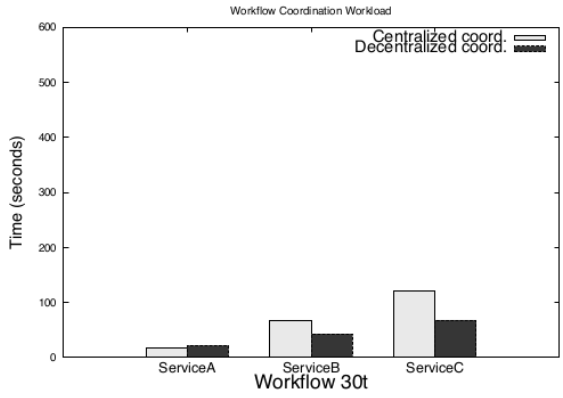

Fig. 19. Coordination workloads for the 30-task graph.

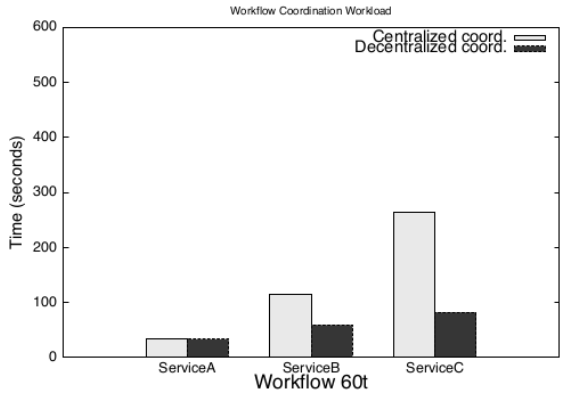

Fig. 20. Coordination workloads for the 60-task graph.

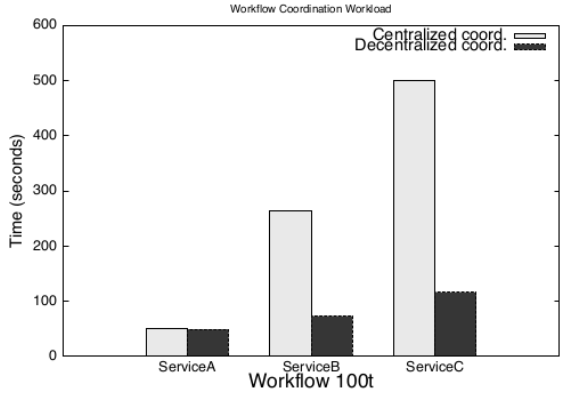

Fig. 21. Coordination workloads for the 100-task graph.

6.3 Exchanging Data

For the second experiment, we have dealt with different amounts of data exchange using the decentralized prototype. We processed six workflows based on the *Workflow30t* graph, whose tasks are bounded to the same web service. For each workflow, we measured the performance using a set of web services exchanging different amounts of data for their execution. This set of services is composed by the three previously-mentioned services and by three others. Thus, experiments were conducted with services exchanging respectively 28, 583, 3053, 5053, 9773, and 15000 bytes of data. The performance obtained accordingly is illustrated in Figure 22.

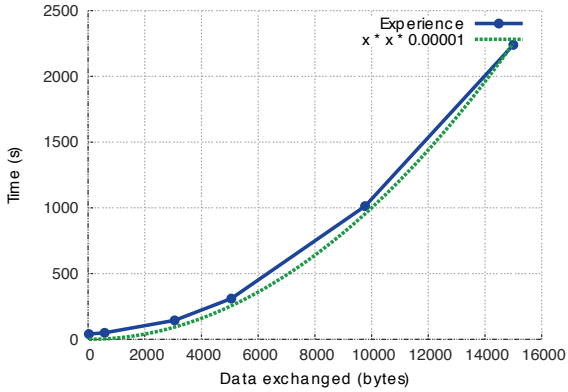

Fig. 22. Performance results, data exchanged.

As we can see in Figure 22, the increase in data exchange among tasks provokes an increase of the execution time, and suggests a clear quadratic behavior of the completion time when the size of the information exchanged increases. Nevertheless, no bottlenecks have been experienced, even if it may appear with higher data rate. The degradation occurs because the information exchanged is considered itself as a molecule in our chemical model, to be transferred and processed in the multiset. Note that, the number of messages exchanged was not measured in the experiments. However, it can be easily deduced by the number of tasks of a workflow (106 messages for the *Workflow30t*). Besides, the number of messages exchanged is the same for both centralized and decentralized systems.

6.4 Discussion

This series of experiments, by offering a proof of concept of the model, while showing its viability in actual deployments, highlights the benefits of a decentralized (chemistry-based) workflow system. Our decentralized workflow engine processes large workflows with a reduced coordination overhead in comparison with a centralized engine. Using the decentralized engine, the coordination is executed locally on each ChWS, instead of being managed by a single node. Furthermore, as shown in our previous work [13], our decentralized engine is also competitive when running real scientific applications in comparison with most common and used workflow management systems, as well as against other decentralized workflow approaches.

However, there are two limitations that come up when using our decentralized workflow system: *i)* the network latency causes performance degradations, which are emphasized when processing workflows having a reduced computational load such as the *Workflow30t*; *ii)* the multiset could constitute a bottleneck. It remains a centralized space shared by every ChWSes leading to potential scalability issues. Following this idea, our approach may experience some performance bottleneck when the rate of data exchange among services becomes very high. The decentralization of the multiset itself was recently addressed through the formulation of solutions based on peer-to-peer protocols, able to distribute and retrieve objects (here, molecules) at large-scale [15]. One of the next steps of this work is to build our current decentralized prototype on top of such approaches to remove the bottleneck problem, and proposes a fully decentralized workflow engine.

Recall, beyond performance or optimization considerations, that the chemical models provide all the needed abstractions to naturally express both data-driven and complex control-driven execution, including particular features like cancellation. Please refer to [22] for more details. We consider the chemical abstraction as participating in the long term objective of improving the workflow execution models on emerging platform, like clouds, where the elasticity brings new modeling challenges.

7 Related Works

There is a vast literature related to the distributed execution of workflows. We observed two methods of distributed coordination approach. In the first one, nodes interact directly. In the second one, they use a shared space for coordination.

Earlier works proposed decentralized architectures where nodes achieve the coordination of a workflow through the exchange of messages [23], [24]. Some works, such as [25], [26], [27], shown the increasing interest in this type of coordination mechanism. In [25], the authors introduce service invocation triggers, a lightweight infrastructure that routes messages directly from a producing service to a consuming one, where each service invocation trigger corresponds to the invocation of a service. In [26], an engine is proposed based on a peer-to-peer application architecture wherein nodes (similar to local engines) are distributed across multiple computer systems, but appear to the users as a single entity. These nodes collaborate, in order to execute a composite Web service with every node executing a part of it. Lately, a continuation-passing style, where information on the remainder of the execution is carried in messages, has been proposed [27]. Nodes interpret such messages and thus conduct the execution of services without consulting a centralized engine. However, this coordination mechanism implies a tight coupling of services in terms of spatial and temporal composition. Nodes need to know explicitly which other nodes they will potentially interact with, and when, to be active at the same time. Likewise, a distributed workflow system based on mobile libraries playing the role of engines was presented in [28]. The authors, however, do not give much details about the coordination itself, and where the data and control dependencies are located.

Our works deal with the information exchange among ChWSes by writing and reading the multiset. Then, the communication can be completely asynchronous since the multiset guarantees the persistence of data and control dependencies. This makes our approach more relevant in a loosely-coupled services environment, and able to deal with dynamic changes in the workflow (as the workflow itself can be rewritten in the multiset).

Another series of works rely on a shared space to exchange information between nodes of a decentralized architecture, more specifically called a *tuplespace* [29], [30], [31], [32]. Its origin can be found in the coordination data-driven languages such as Linda [33], as a parallel programming extension for programming languages for the purpose of separating coordination logic from program logic. Linda builds upon the notion of a tuplespace, which is a piece of memory shared by all interacting parties. Using a tuplespace for coordination, the execution of a part of a workflow within each node is triggered when tuples, matching the templates registered by the respective nodes, are present in the tuplespace. In the same vein, works such as [34], propose a distributed architecture based on Linda where distributed tuplespaces store data and programs as tuples, allowing mobile computations by transferring programs from one tuple to another. However, the chemical paradigm allows an increased abstraction level while providing support for dynamics.

Based on this coordination method, works such as [29], [30] and [31] replace a centralized BPEL engine by a set of distributed, loosely coupled, cooperating nodes. In [29] and [30], the authors present a coordination mechanism where the data is managed using a tuplespace and the control is driven by asynchronous messages exchanged between nodes. This message exchange pattern for the control is derived from a Petri net expression of the workflow. However, while in these works, the tuplespace is only used to store data information, our coordination mechanism stores both control and data information in the multiset, which is made possible by the use of the chemical execution model for the coordination of all data and control dependencies.

As a continuation of [30], [32] designed and implemented a tuplespace-based process execution middleware that transforms a workflow definition into a set of activities. Using this system, control and data dependencies are now stored in the tuplespace. Activities are distributed by passing tokens in the Petri net that formalize of the tuplespace-based interactions. The most significant difference with the present work, however, is the general goal. The authors focused on the process definition and the viability of its prototype, while its experimental validation remains a wide open issue, in particular related to scalability. Also, they do not discuss the advantages and disadvantages of decentralization in workflow processing. Similarly, based on the theoretical underpinnings and core of the engine presented in [32], the work in [31] uses a shared tuplespace working as a communication infrastructure, allowing to exchange control and data dependencies among processes and make the different nodes interact. The authors transform a centralized BPEL definition into a set of coordinated processes using again the tuplespace as a

communication medium. However, the use of BPEL as the coordination language hinders from expressing dynamic and self-adaptive behaviors.

As a more general comment, to our knowledge, these works do not provide any experimental validation of running workflows on distributed infrastructures.

8 Conclusion

Most of today's approaches to the coordination of composite Web services are based on highly centralized architectures. Such systems present several drawbacks, mainly dealing with scalability, fault-tolerance, and privacy. In order to tackle these issues, it becomes today crucial to propose decentralized coordination mechanisms. However, current proposals for decentralized workflow coordination require tight coupling of services, and use workflow description languages that do not provide concepts for distributed workflow execution. In this paper, we have proposed a high-level coordination mechanism allowing a distributed execution of composite Web services, based on the chemical metaphor. Our chemical programming paradigm expresses parallelism and autonomic behaviors naturally using a higher-order chemical language. We have introduced the notion of chemical Web service, which encapsulates a Web service. Through a shared multiset containing the information on both data and control dependencies needed for coordination, chemical Web services are co-responsible for carrying out the execution of a workflow in the composite services in which they appear. Spatial and temporal composition of services is achieved dynamically through this shared multiset. Their coordination is decentralized and distributed among individual Web service's chemical engine executing a part of the workflow. Through the deployment of a software prototype following these concepts, and its experimental validation over an actual platform, we have been able to provide a proof of concept while showing its viability and identifying its performance limitations for its future improvement.

References

[1] M. Rosen, B. Lublinsky, K. T. Smith, and M. J. Balcer, *Applied SOA: Service-Oriented Architecture and Design Strategies*. Wiley, Jun. 2008.

[2] "Amazon's outage in third day: debate over cloud computing's future begins." [Online]. Available: http://venturebeat.com/2011/04/23/amazons-outage-in-third-day-debate-over-cloud-computings-future-begins/

[3] G. Alonso, D. Agrawal, A. E. Abbadi, and C. Mohan, "Functionality and limitations of current workflow management systems," *submitted to IEEE Expert*, 1997. [Online]. Available: http://www.almaden.ibm.com/cs/exotica/wfmsys.pdf

[4] Y. Yang, "An architecture and the related mechanisms for web-based global cooperative teamwork support," *Int. Journal of Computing and Informatics*, vol. 24, 2000.

[5] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda, "Decentralized orchestration of composite web services," *In Proceedings of the 13th International World Wide Web Conference, (WWW2004)*, pp. 134–143, 2004.

[6] M. Viroli and F. Zambonelli, "A biochemical approach to adaptive service ecosystems," *Information Sciences*, pp. 1–17, 2009.

[7] J.-P. Banâtre, T. Priol, and Y. Radenac, "Chemical Programming of Future Service-oriented Architectures," *Journal of Software*, vol. 4, no. 7, pp. 738–746, 2009.

[8] J. Banâtre, P. Fradet, and Y. Radenac, "Generalised multisets for chemical programming," *Mathematical Structures in Computer Science*, vol. 16, no. 4, pp. 557–580, 2006.

[9] Z. Németh, C. Pérez, and T. Priol, "Distributed workflow coordination: molecules and reactions," in *20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[10] H. Fernandez, T. Priol, and C. Tedeschi, "Decentralized approach for execution of composite web services using the chemical paradigm," in *IEEE International Conference on Web Services (ICWS)*, 2010, pp. 139–146.

[11] J.-P. Banâtre and D. L. Métayer, "The gamma model and its discipline of programming," *Sci. Comput. Program.*, vol. 15, no. 1, pp. 55–77, 1990.

[12] "Web services business process execution language, (WS-BPEL), Version 2.0," OASIS Standard, 2007.

[13] H. Fernandez, "Flexible Coordination based on the Chemical Metaphor for Service Infrastructures," THESE, Université Rennes 1, Jun. 2012. [Online]. Available: http://tel.archives-ouvertes.fr/tel-00717057

[14] J. Protić, M. Tomasević, and V. Milutinović, *Distributed shared memory*. John Wiley and Sons, 1998.

[15] M. Obrovac and C. Tedeschi, "Deployment and evaluation of a decentralised runtime for concurrent rule-based programming models," in *(ICDCN)*, 2013, pp. 408–422.

[16] H. Fernandez, C. Tedeschi, and T. Priol, "Decentralized workflow coordination through molecular composition," in *ICSOC Workshops*, 2011, pp. 22–32.

[17] J. Recker, "BPMN modeling – who, where, how and why," *BP-Trends*, vol. 5, no. 5, pp. 1–8, 2008.

[18] Y. Radenac, "Programmation "chimique" d'ordre supérieur," Thèse de doctorat, Université de Rennes 1, 2007.

[19] P. Leitner, F. Rosenberg, and S. Dustdar, "Daios: Efficient dynamic web service invocation," *IEEE Internet Computing*, vol. 13, no. 3, pp. 72–80, 2009.

[20] R. Bolze, F. Cappello, E. Caron, M. J. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, B. Quétier, O. Richard, E.-G. Talbi, and I. Touche, "Grid'5000: A large scale and highly reconfigurable experimental grid testbed," *IJHPCA*, vol. 20, no. 4, pp. 481–494, 2006.

[21] G. Berriman, E. Deelman, J. Good, J. Jacob, D. Katz, C. Kesselman, A. Laity, T. Prince, G. Singh, and M. hu Su, "Montage: A grid enabled engine for delivering custom science-grade mosaics on demand," *In Proceedings of SPIE Conference 5487: Astronomical Telescopes*, 2004.

[22] H. Fernández, C. Tedeschi, and T. Priol, "Self-coordination of Workflow Execution Through Molecular Composition," INRIA, Research Report RR-7610, 05 2011. [Online]. Available: http://hal.inria.fr/inria-00590357/PDF/RR-7610.pdf

[23] J. Yan, Y. Yang, and G. Raikundalia, "Enacting business processes in a decentralised environment with p2p-based workflow support," in *Advances in Web-Age Information Management*, 2003, pp. 290–297.

[24] M. G. Nanda, S. Chandra, and V. Sarkar, "Decentralizing execution of composite web services," in *Proc. of the 19th Conf. on object-oriented programming, systems, languages, and applications*. ACM, 2004, pp. 170–187.

[25] W. Binder, I. Constantinescu, and B. Faltings, "Decentralized orchestration of compositeweb services," in *Proc. of the IEEE Int. Conference on Web Services*. IEEE Computer Society, 2006, pp. 869–876.

[26] R. A. Micillo, S. Venticinque, N. Mazzocca, and R. Aversa, "An agent-based approach for distributed execution of composite web services," in *IEEE International Workshops on Enabling Technologies*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 18–23.

[27] W. Yu, "Consistent and decentralized orchestration of BPEL processes," in *Proceedings of the 2009 ACM symposium on Applied Computing*. Honolulu, Hawaii: ACM, 2009, pp. 1583–1584.

[28] P. Downes, O. Curran, J. Cunniffe, and A. Shearer, "Distributed radiotherapy simulation with the webcom workflow system," *Int. Journal of High Performance Computing Applications*, vol. 24, p. 213–227, 2010.

[29] P. A. Buhler and J. M. Vidal, "Enacting BPEL4WS specified workflows with multiagent systems," *In Proceedings of the Workshop on Web Services and Agent-Based Engineering*, 2004.

[30] D. Martin, D. Wutke, and F. Leymann, "A novel approach to decentralized workflow enactment," in *IEEE International Enterprise Distributed Object Computing Conference*. Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 127–136.

[31] M. Sonntag, K. Gorlach, D. Karastoyanova, F. Leymann, and M. Reiter, "Process space-based scientific workflow enactment," *Int. Journal of Business Process Integration and Mmgt*, vol. 5, pp. 32 – 44, 2010.

[32] D. Martin, "A tuplespace based execution model for decentralized workflow enactment : applied for BPEL," Ph.D. dissertation, University of Stuttgart , Germany, Sep. 2010.

[33] D. Gelernter and N. Carriero, "Coordination languages and their significance," *Commun. ACM*, vol. 35, no. 2, pp. 96–107, 1992.
[34] R. D. Nicola, G. Ferrari, and R. Pugliese, "KLAIM: a kernel language for agents interaction and mobility," *IEEE Transactions On Software Engineering*, vol. 24, 1997.

Hector Fernandez Hector Fernandez is a Researcher Scientist member of the ConPaaS team at Vrije University of Amsterdam. Dr. Fernandez received his PhD in Computer Science from University of Rennes 1 in 2012 in the area of service-oriented computing. During his PhD, his research interests included service composition and service coordination in distributed systems, and in particular workflow management systems. Nowadays, he continues research on service infrastructures by designing an auto-scaling system for an open-source cloud platform called ConPaaS.

Cédric Tedeschi Cédric Tedeschi is an Assistant Professor at the University of Rennes 1 and holds a position at the IRISA laboratory. He is a member of the Myriads project hosted by INRIA. He received a PhD in Computer Science from the Ecole normale supérieure de Lyon in 2008. His current research interests include coordination in distributed systems and programming models for service infrastructures.

Thierry Priol Thierry Priol is a research director at Inria. He joined Inria in 1989. He was Director of the ACI GRID of the Ministry of Research and the Scientific Coordinator of the CoreGRID network of excellence funded by the European Commission. From 1999 to 2009, he was a project-team leader at the Inria Rennes Bretagne Atlantique centre. He has conducted his research in the field of parallel computing, image synthesis algorithm parallelisation, and programming tools based on the concept of shared virtual memory. He has also worked on the programming of computing grids, proposing extensions to software component models for code coupling applications.